Article
Brendan Bannon · Dec 27, 2016 8m read

The Art of Mapping Globals to Classes (4 of 3)

The Art of Mapping Globals to Classes (4 of 3)

The forth in the trilogy, anyone a Hitchhikers Guide to the Galaxy fan?

If you are looking to breathe new life into an old MUMPS application follow these steps to map your globals to classes and expose all that beautiful data to Objects and SQL.

If the above does not sound familiar to you please start at the beginning with the following:

The Art of Mapping Globals to Classes (1 of 3)

The Art of Mapping Globals to Classes (2 of 3)

The Art of Mapping Globals to Classes (3 of 3)

This one is for you Joel! Building on the parent-child relationship we defined in the last example, here we are going to create a grandchild class to handle the new seasons info added to the ^ParentChild Global.

Same disclaimer: If you can ' t make heads or tails of your globals after reviewing these articles please contact the WRC and we will try to help you out: Support@InterSystems.com.

Steps for Mapping a Global to a Class:

1. Identify a repeating pattern in the global data.
2. Identify what makes up a unique key.
3. Identify the properties and their types.
4. Define the properties in the class (don ' t forget the properties from the variable subscripts).
5. Define the IdKey index.
6. Define the Storage Definition:
    a. Define the Subscripts up to and including the IdKey.
    b. Define the Data section.
    c. Ignore the Row ID section. 99% of the time the default is what you want so let the system fill that in.
7. Compile and test your class / table.

^ParentChild(1)="Brendan^45956"

^ParentChild(1,"Hobbies",1)="Pit Crew"

^ParentChild(1,"Hobbies",1,"Seasons")="Fall*Winter"

^ParentChild(1,"Hobbies",2)="Kayaking"

^ParentChild(1,"Hobbies",2,"Seasons")="Spring*Summer*Fall"

^ParentChild(1,"Hobbies",3)="Skiing"

^ParentChild(1,"Hobbies",3,"Seasons")="Summer*Winter"

^ParentChild(2)="Sharon^46647"

^ParentChild(2,"Hobbies",1)="Yoga"

^ParentChild(2,"Hobbies",1,"Seasons")="Spring*Summer*Fall*Winter"

^ParentChild(2,"Hobbies",2)="Scrap booking"

^ParentChild(2,"Hobbies",2,"Seasons")="Spring*Summer*Fall*Winter"

^ParentChild(3)="Kaitlin^56009"

^ParentChild(3,"Hobbies",1)="Lighting Design"

^ParentChild(3,"Hobbies",1,"Seasons")="Spring*Summer*Fall*Winter"

^ParentChild(3,"Hobbies",2)="pets"

^ParentChild(3,"Hobbies",2,"Seasons")="Spring*Summer*Fall*Winter"

^ParentChild(4)="Melissa^56894"

^ParentChild(4,"Hobbies",1)="Marching Band"

^ParentChild(4,"Hobbies",1,"Seasons")="Fall"

^ParentChild(4,"Hobbies",2)="Pep Band"

^ParentChild(4,"Hobbies",2,"Seasons")="Winter"

^ParentChild(4,"Hobbies",3)="Concert Band"

^ParentChild(4,"Hobbies",3,"Seasons")="Spring*Summer*Fall*Winter"

^ParentChild(5)="Robin^57079"

^ParentChild(5,"Hobbies",1)="Baking"

^ParentChild(5,"Hobbies",1,"Seasons")="Spring*Summer*Fall*Winter"

^ParentChild(5,"Hobbies",2)="Reading"

^ParentChild(5,"Hobbies",2,"Seasons")="Spring*Summer*Fall*Winter"

^ParentChild(6)="Kieran^58210"

^ParentChild(6,"Hobbies",1)="SUBA"

^ParentChild(6,"Hobbies",1,"Seasons")="Summer"

^ParentChild(6,"Hobbies",2)="Marching Band"

^ParentChild(6,"Hobbies",2,"Seasons")="Fall"

^ParentChild(6,"Hobbies",3)="Rock Climbing"

^ParentChild(6,"Hobbies",3,"Seasons")="Spring*Summer*Fall"

^ParentChild(6,"Hobbies",4)="Ice Climbing"

^ParentChild(6,"Hobbies",4,"Seasons")="Winter"

Step 1:

Well it is not too hard to find the repeating data for our new class, it is the Seasons sub node. The tricky part comes when I say I don ' t want " Spring*Summer*Fall " to all be in one row, I want 3 rows: " Spring " , " Summer " , " Fa

^ParentChild(1)="Brendan^45956"

^ParentChild(1,"Hobbies",1)="Pit Crew"

^ParentChild(1,"Hobbies",1,"Seasons")="Fall*Winter"

^ParentChild(1,"Hobbies",2)="Kayaking"

^ParentChild(1,"Hobbies",2,"Seasons")="Spring*Summer*Fall"

^ParentChild(1,"Hobbies",3)="Skiing"

^ParentChild(1,"Hobbies",3,"Seasons")="Summer*Winter"

Each hobby can have 1 to 4 seasons. I guess we could create 4 more properties in the Example3Child class, but what would we do if someone invents a new season? A more flexible solution would be to create a grandchild table, so the number of seasons can remain dynamic.

Step 2:

We all know to look at the subscripts to get the parts of the IdKey, so we know Subscript 1 and Subscript 3 will be part of the IdKey, we need one more value to uniquely identify the different seasons, but we are out of subscripts!

The information we are putting in the mapping is used to generate code for queries. Maybe if we think about what COS commands are needed to get this info, it will help us define the mapping. The 3 big things we need to do are:

```
SET sub1=$ORDER(^Parentchild(sub1))

SET sub2=$ORDER(^Parentchild(sub1, " Hobbies " ,sub2))

SET season=$PIECE(^ParentChild(sub1, " Hobbies " ,sub2 " , " Seasons " ), " * " ,PC)
```

We can do the same thing in the Subscripts section on the mapping. Caché SQL Storage supports 4 different types of Subscripts: Piece, Global, Sub, and Other. The default is Sub and that is what we have been using so far, which gets us those lovely $ORDER() loops we need.

In this example we will introduce the Piece options. The property that you use at this level will be used as a Piece Counter (PC in the example above). The default behavior is for it to increment this by 1, until we get to the end of the string.

Step 3:

3 properties: the data is the easy: Season, then we have the relationship property: HobbyRef, and finally we need a childsub: PieceCount

Step 4:

Property Season As %String;

Property PieceCounter As %Integer;

Relationship HobbyRef As Mapping.Example3Child [Cardinality = parent, Inverse = Seasons];

Step 5:

When we look at the Subscripts mapping we will have 3 variable levels, but for the IdKey index we only refer to 2 properties HobbyRef and PieceCounter

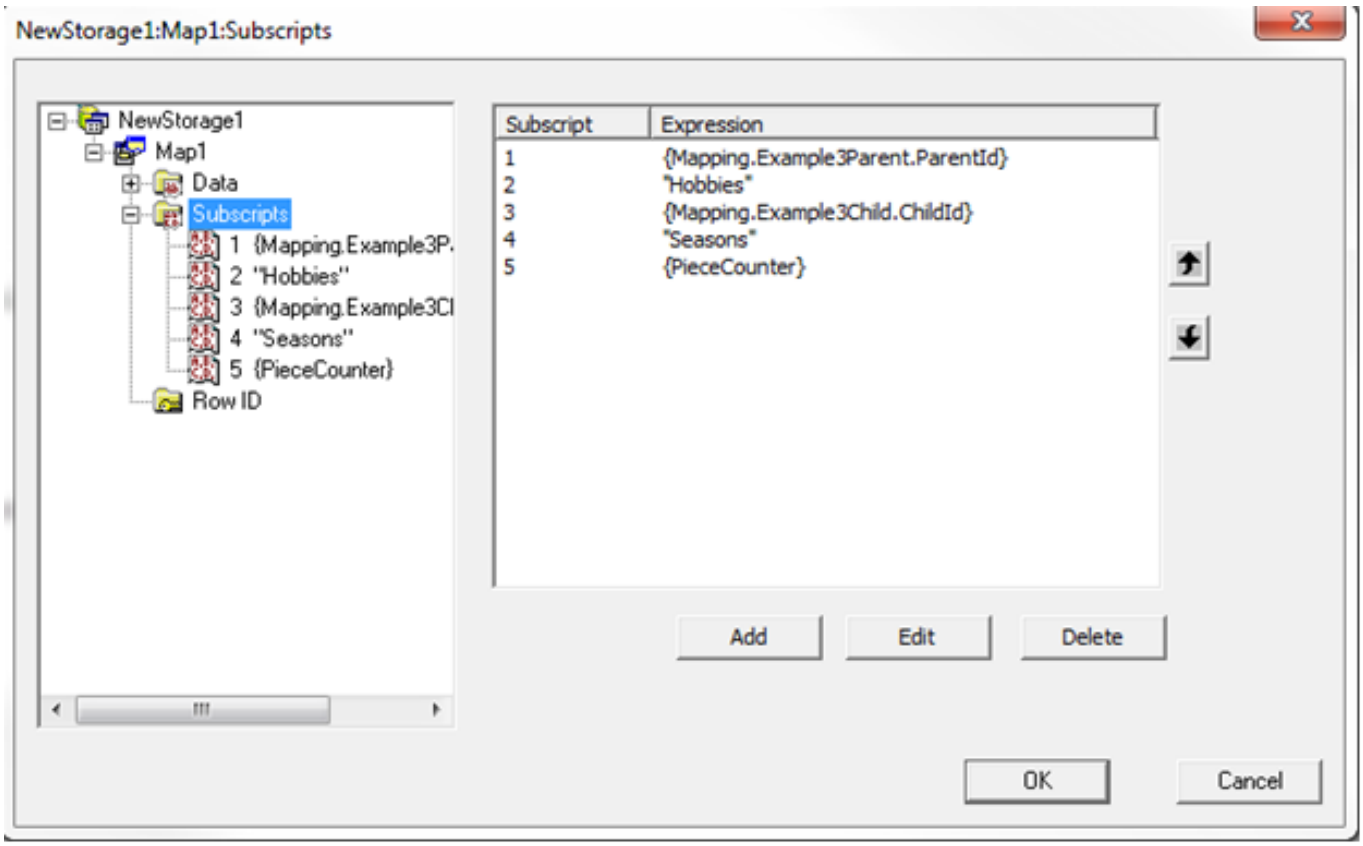
Index Master On (HobbyRef, PieceCounter) [IdKey];

Step 6:

Same three sections we have been working with all along. We are still ignoring Row ID. For this one, we need to go into a little more detail for the Subscripts and define the Access Type. This class will use ‘ Sub ’ and ‘ Piece ’. If you want to see examples of ‘ Global ’ and ‘ Other ’ you will need to get your hands on my zip file of examples.

Step 6a:

The main Subscripts page looks the same as always, there are just two more levels added. Remember for the two parts of the Parent reference we need to refer back to the classes that we reference: {Mapping.Example3Parent} and {Mapping.Example3Child}. The difference shows up when you click on one of the Subscript Levels on the left side of the window.



In the image below you can see all the different scary things you can do at each Subscript Level. For Access Type of ‘ Sub ’ we assume you want to start at “ ” and $ORDER() until you get to “ ”. If that is not the case you can provide a Start Value and or a Stop Value.

‘ Data Access ’ lets you change what you are looking at from the previous level, for example, you could change the global you want to loop over.

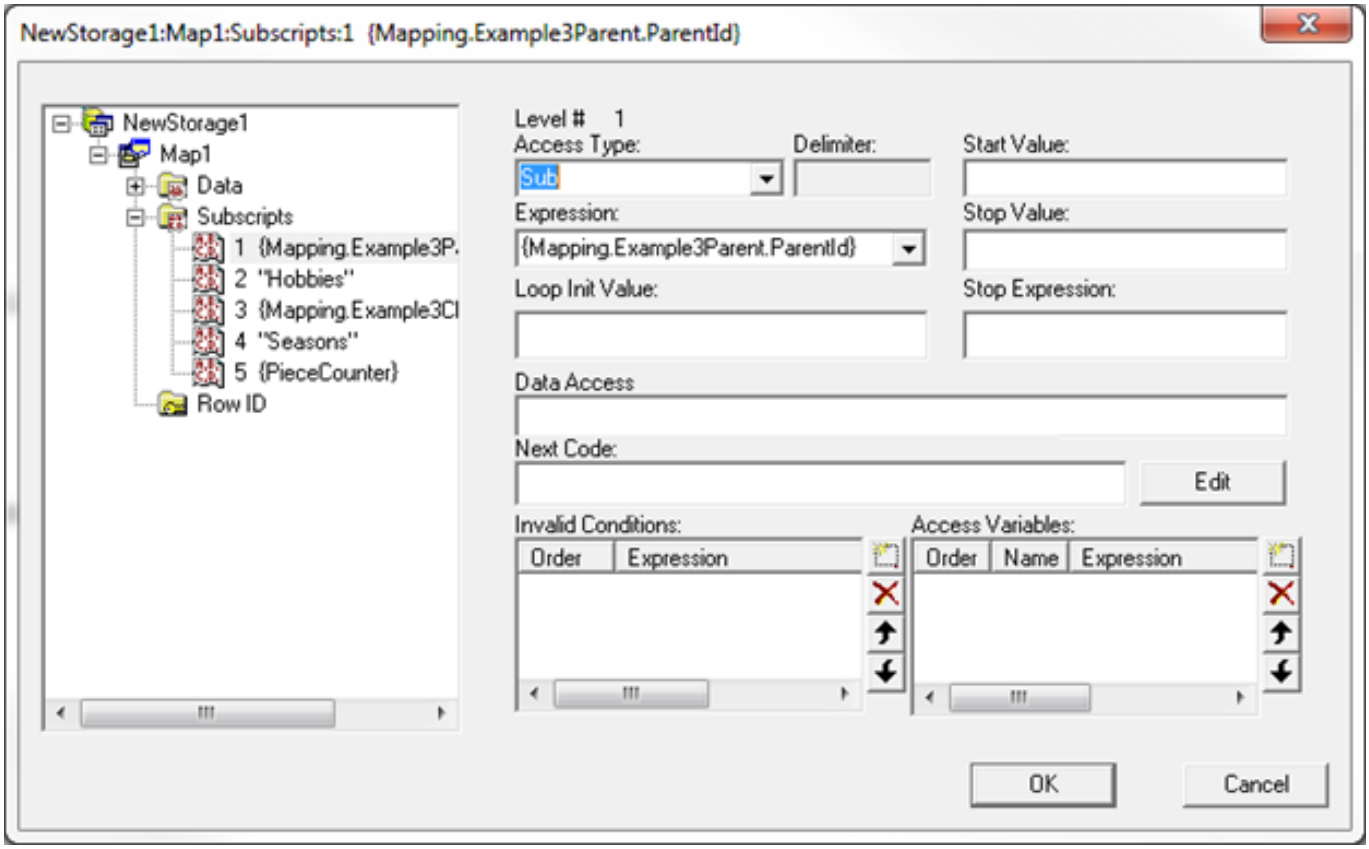
‘ Next Code ’ and ‘ Invalid Conditions ’ go together, and are the most common thing you will use in this window. If a simple $ORDER() will not get you from one valid value to the next you can write your own code for us to use instead.

‘ Next Code ’ would be used to get from one good value to the next good value, just like a $ORDER() does.

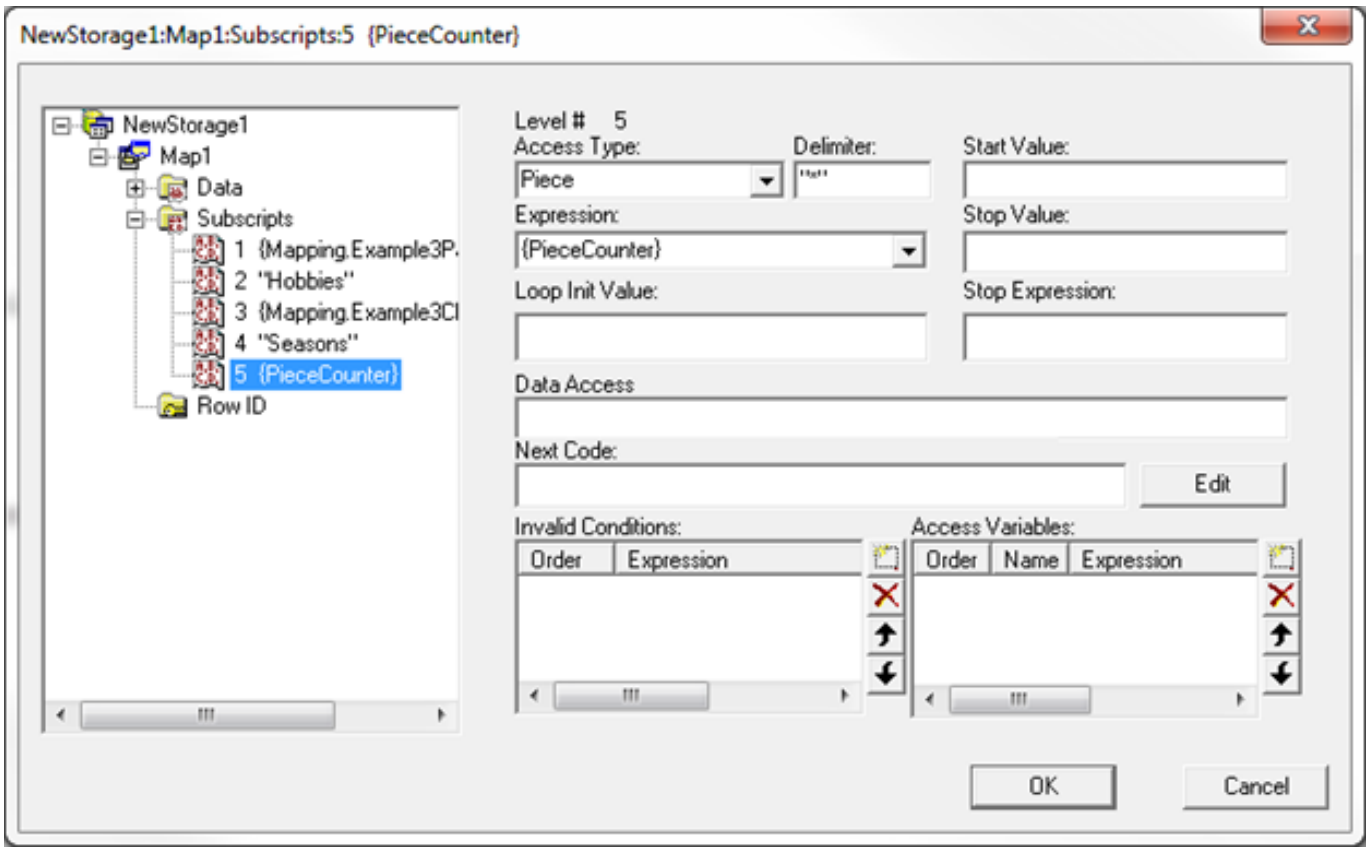
The ‘ Invalid Condition ’ is use for testing a given value. If you provided a value for ‘ subscriptX ’ there is no need to call the Next to find it, you provided it already. What is needed, is some code that will figure out if the value is good or not.

My long promised zip file of examples has lots of classes that use ‘ Next Code ’ and ‘ Invalid Conditions ’.

‘ Access Variables ’ are the last thing on the page, and are rarely used. Basically, you can setup a variable here, give it a value in one Subscript Level, then use it in higher Subscript Levels. The generated table code will deal with the scoping for you.

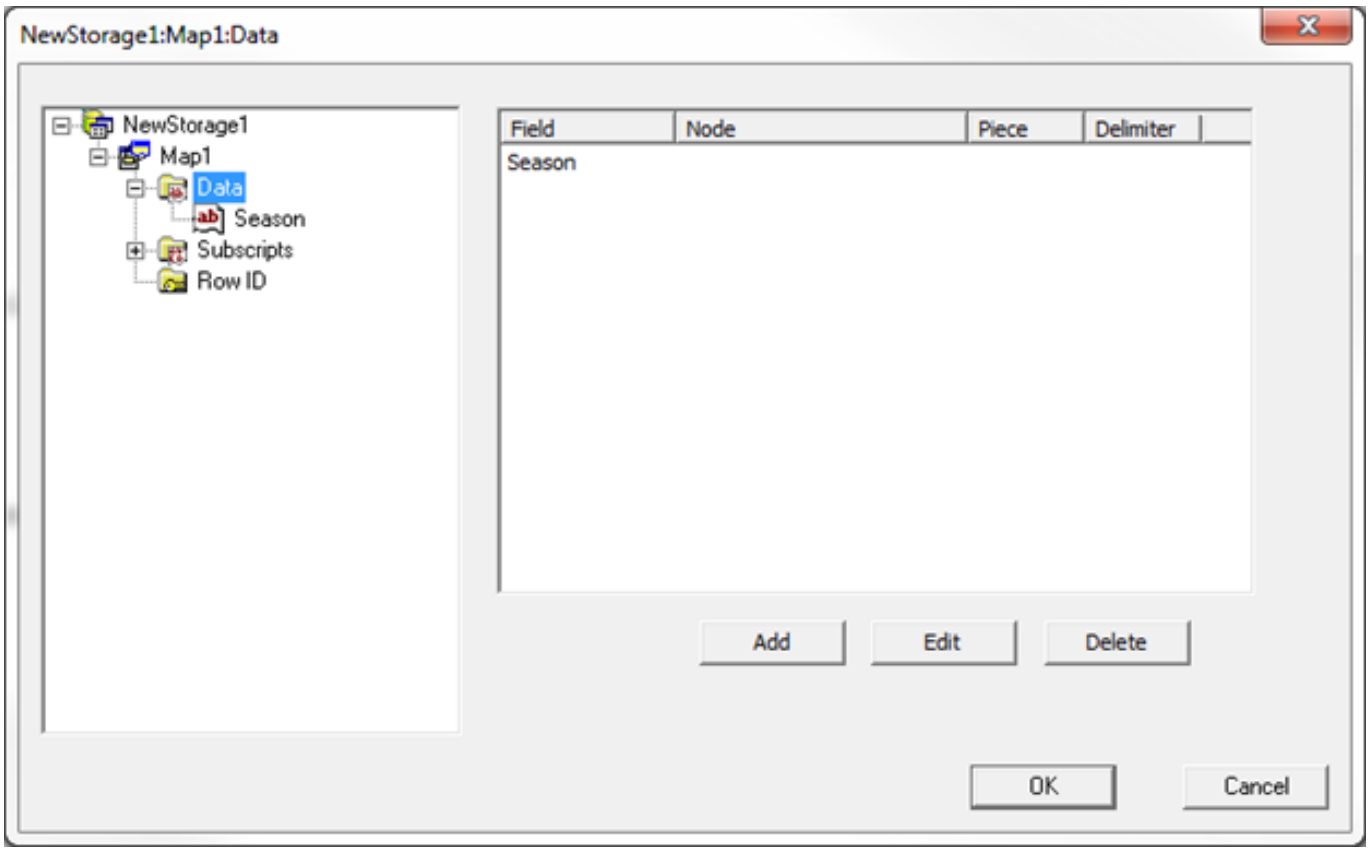


For Subscript Level 5 the ‘ Access Type ’ is ‘ Piece ’ and the ‘ Delimiter ’ is “ * ”. The generated code will start at Piec and increment by 1 until we run out of $PIECE values. Again, we could provide Start Values, and or Stop Values to control this.



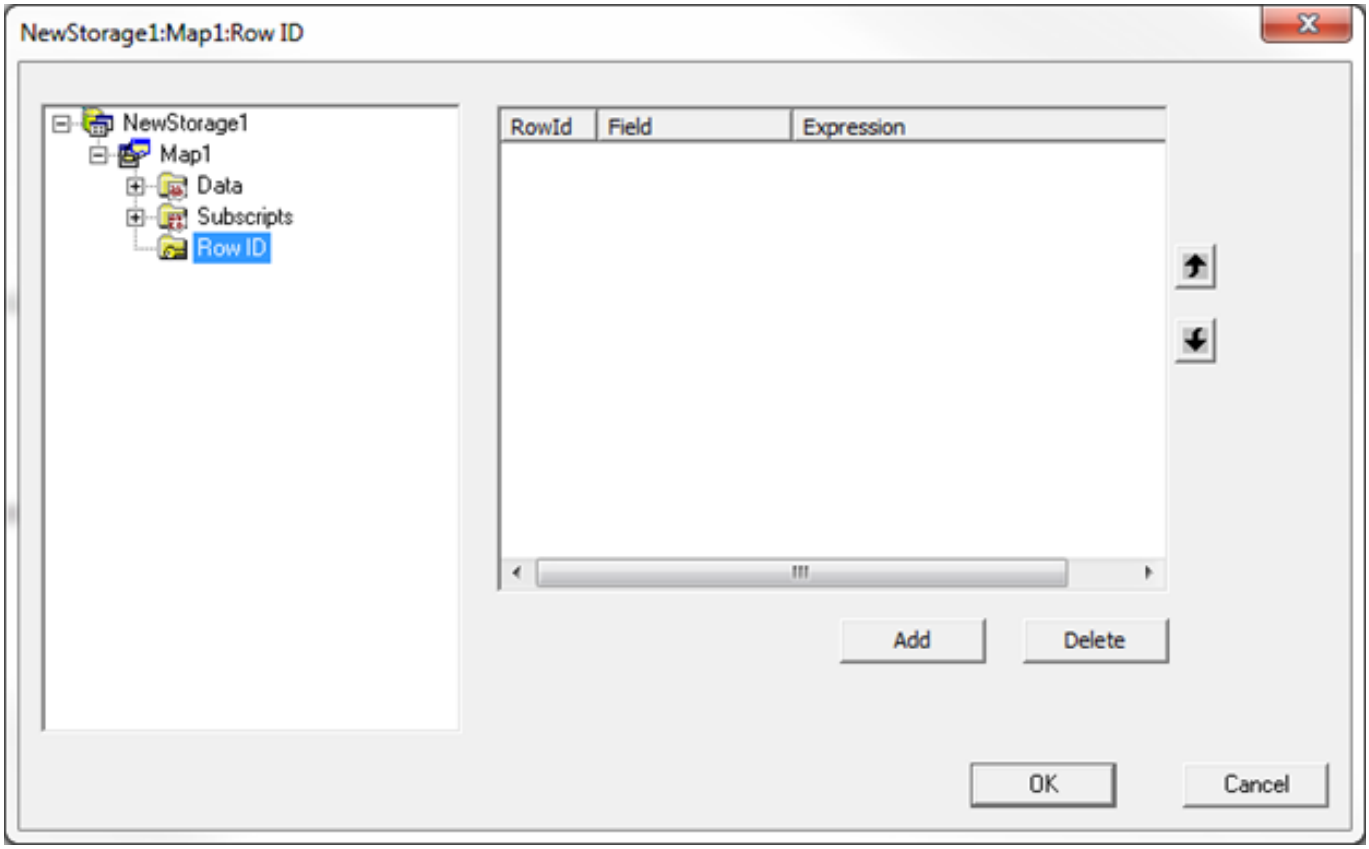
Step 6b:

The data section is just one property; no need for a ‘ Piece ’ or ‘ Delimiter ’ . If this table had more fields, we most likely would need to provide a ‘ Piece ’ and ‘ Delimiter ’ and that would be fine.



Step 6c:

Still leaving this blank.



Step 7:

Everything compiles nice and clean.

```
Compilation started on 11/30/2016 08:17:42 with qualifiers 'uk/importselectivity=1 /checkuptodate=expandedonly'
Compiling 2 classes, using 2 worker jobs
Compiling class Mapping.Example3Child
Compiling class Mapping.Example3GrandChild
Compiling table Mapping.Example3GrandChild
Compiling table Mapping.Example3Child
Compiling routine Mapping.Example3Child.1
Compiling routine Mapping.Example3GrandChild.1
Compilation finished successfully in 1.021s.
```

A join of the three tables:

```
SELECT P.ID, P.Name, P.DateOfBirth,

C.ID, C.Hobby, G.ID, G.Season

FROM Mapping.Example3Parent P

JOIN Mapping.Example3Child C ON P.ID = C.ParentRef

JOIN Mapping.Example3Grandchild G ON C.ID = G.HobbyRef

WHERE P.Name = 'Kieran'
```

Gives us:

ID	Name	DateOfBirth	ID	Hobby	ID	Season
6	Kieran	05/16/2000	6¦¦1	SUBA	6¦¦1¦¦1	Summer
6	Kieran	05/16/2000	6¦¦2	Marching Band	6¦¦2¦¦1	Fall
6	Kieran	05/16/2000	6¦¦3	Rock Climbing	6¦¦3¦¦1	Spring
6	Kieran	05/16/2000	6¦¦3	Rock Climbing	6¦¦3¦¦2	Summer
6	Kieran	05/16/2000	6¦¦3	Rock Climbing	6¦¦3¦¦3	Fall
6	Kieran	05/16/2000	6¦¦4	Ice Climbing	6¦¦4¦¦1	Winter

Remember, I said the child table IdKey is always made up of 2 parts: The Parent reference and the Childsub.

In the first row the Example3Child ID is 6¦¦1: Parent Reference = 6 and Childsub = 1.

For Example3GrandChild the IdKey is made up of three parts 6¦¦1¦¦1, but it is still just the Parent Reference and the Childsub. The Parent reference is just getting a little more complex: Parent Reference = 6¦¦1 and Childsub = 1.

In Example3Child, the number of properties in the subscripts matches the number of Properties in the IdKey. As we nest deeper in a parent-child structure the IdKey becomes compound and the number of subscripts will increase.

Here is an export of the 3 classes used in this example: MappingExample4.zip.

#Globals #Mapping #Object Data Model #SQL #Caché